CIS 130 - Homework #8

Due:

Thursday, April 29th, 11:59 pm

Purpose:

Mostly practice writing C++ functions and programs involving repetition and arrays

How to submit:

When you are done with the following problems:

- (Assuming that you are still ssh'd and logged into to **nrs-labs**, since you will be writing and testing some of these C++ functions on nrs-labs using the `funct_play2/funct_compile/ expr_play` tools, and writing and testing others using `nano` and `g++`)
- Make sure your current working directory on **nrs-labs** is the one where your homework files are: do the command:

  ```
  ls
  ```

  ...and make sure you see the names of your homework files listed.
  - If you are not in the proper directory, use `cd` *`directory_name`* to go to the proper directory. (For example, `cd 130hw08`)
- use `~st10/130submit` to submit all of your `.cpp` and `.h` files in the current directory.
  - Remember, I don't mind if extra `.cpp` and `.h` files get submitted as well.
  - Make sure that `~st10/130submit` shows that it submitted all of your homework files.
- (ASK ME if this is not clear, or if you have any problems with submission!)

Important notes:

- Each student should work individually on this assignment.
- Remember, you are still expected to follow the Design Recipe for all functions that you write. (Remember to use C++ type names in C++ function contracts, and to write C++ specific examples/tests as discussed in class.)
- Remember to follow the class style guidelines.

Homework problems:

Problem 0

Create, protect, and change to a directory `130hw08` -- type the following from your home directory on nrs-labs:

```
[you1@nrs-labs ~]$ mkdir 130hw08

[you1@nrs-labs ~]$ chmod 700 130hw08
```

```
[you1@nrs-labs ~]$ cd 130hw08
```

(If you log out and come back later, remember to `cd 130hw08` each time to return to this directory!)

Problem 1

Here is some pseudocode, giving the logic needed for a function that expects an array of numbers and its size, and returns the smallest element in that array:

```
set the smallest seen so far to be the first element
while there are more array elements to check
    if the latest array value is less than the smallest
    seen so far,
        make that the new smallest seen so far
    increment my count
return smallest seen so far
```

Using the design recipe, use `funct_play2` to develop a C++ function `get_smallest` that indeed expects an array of numbers and its size, and produces the smallest element in that array.

As in the in-class example `sum_array`:

- be use to include a declarations and initialization for an example array and its size before each specific example call of `get_smallest`
- [style note: give this example array and its size *different* names than your parameter array and its size]

To test `get_smallest`, either use `nano` (or your favorite text editor) to write a small testing `main` function in a file named `get_smallest_test.cpp`, using the `main` function template (`main_template.txt`) available on the public course web page, or edit `get_smallest_ck_expect.cpp` appropriately. In either case, here are the requirements:

- it must declare and initialize at least two arrays of different sizes, using named constants for these sizes
- it must print to the screen a message saying that you are testing `get_smallest`, and that `true`'s mean the tests have passed
- for at least two example calls to `get_smallest`, it should print to the screen the result of comparing that example call to the value it should return (as you see in the posted `sum_array_ck_expect.cpp`)

Submit your files `get_smallest.h`, `get_smallest.cpp`, and either `get_smallest_ck_expect.cpp` or `get_smallest_test.cpp`

Problem 2

Use `funct_play2` to develop a C++ function `how_many` that expects a desired string, an array of strings, and the array's size, and returns the number of time the desired string appears in that array.

(suggestion: during the design recipe, after making your specific examples/tests, develop pseudocode for how you figured out how many times your example string appeared in your example array -- how did you know when a match was found? how do you keep track of how many you've seen so far?)

As in the in-class example `sum_array`:

- be use to include a declarations and initialization for an example array and its size before each specific example call of `how_many`
- [style note: give this example array and its size *different* names than your parameter array and its size]
- (hint: for this one there had better be more than one example call! What are the different cases you should check? But as an additional hint -- if you design it well, you can probably use the same example array for more than one example call. What should be different in those calls, then?)

To test `how_many`, either use `nano` (or your favorite text editor) to write a small testing `main` function in a file named `how_many_test.cpp`, using the `main` function template (`main_template.txt`) available on the public course web page, or edit `how_many_ck_expect.cpp` appropriately. In either case, here are the requirements:

- it must declare and initialize at least one appropriate array, using a named constant for its size
- it must print to the screen a message saying that you are testing `how_many`, and that `true`'s mean the tests have passed
- for an appropriate set of example calls to `how_many`, it should print to the screen the result of comparing that example call to the value it should return (as you see in the posted `sum_array_ck_expect.cpp`)

Submit your files `how_many.h`, `how_many.cpp`, and either `how_many_ck_expect.cpp` or `how_many_test.cpp`.

Problem 3

Consider `line_of_X` from Homework 7. You could use this to create a kind of horizontal bar chart, calling it for each of a set of values. And what is an array but a set of values?

Using `funct_play2`, develop a C++ function `bar_chart` that expects an array of integers and its size, and doesn't return anything, but has the side-effect of printing to the screen a horizontal bar chart with the help of `line_of_X`, printing a line of X's the length of each array value. This function must appropriately call `line_of_X`.

That is, the fragment:

```
const int NUM_MSRS = 7;
int measures[NUM_MSRS] = {3, 1, 6, 2, 8, 4, 5};
bar_chart(measures, NUM_MSRS);
```

...would return nothing, but would have the side-effect of causing the following to be printed to the screen:

```
XXX
X
XXXXXX
```

```
XX
XXXXXXXX
XXXX
XXXXX
```

Because `bar_chart` is a `void` function, we need to write our own small main function to try out this void function. Using `nano` (or your favorite text editor), write a `main` function in a file named `bar_chart_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template.

`bar_chart_test.cpp` must:

- call `bar_chart` at least 2 times, each time with different arrays of different sizes
- precede each `bar_chart` call with a `cout` describing what the user should see (so someone looking at just the results on-screen can tell if they are reasonable)

Submit your files `bar_chart.h`, `bar_chart.cpp`, and `bar_chart_test.cpp`.

Problem 4

It might be interesting to have a way to fill an array with some rather-flaky random integers.

Consider the example function `guess` from class -- it generated a rather-flaky random number for a simple guessing-game.

Using `funct_play2`, develop a C++ function `fill_random` that expects an array of integers, its size, and a maximum desired value, and it doesn't return anything, but it does have the side-effect of filling that array's elements with rather-flaky random integers in the interval [1, that given desired maximum value].

That is, the fragment:

```
const int NUM_COUNTS = 10;
const int MAX_DESIRED = 65;
int counts[NUM_COUNTS];
fill_random(counts, NUM_COUNTS, MAX_DESIRED);
```

...would return nothing, but would have the side-effect of causing array `counts` to now contain `NUM_COUNTS` elements each in the interval [1, `MAX_DESIRED`].

IMPORTANT NOTE: That odd little `srand( time(NULL) );` statement in `guess.cpp` is important; it seeds the random number generator, and by seeding it with the current time, you get slightly different flaky-random numbers each time you call it, as long as those calls are not too close together. But you only want to call it ONCE in a single program, and then call `rand()` as many times as you like in that program.

...That is, be sure to call `srand` only ONCE in `fill_random`, before its loop...

Because `fill_random` is a `void` function, we need to write our own small main function to try out this void function. Using `nano` (or your favorite text editor), write a `main` function in a file named `fill_random_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template.

`fill_random_test.cpp` must:

- call `fill_random` at least 2 times, each time with different arrays of different sizes
- to conveniently see `fill_random`'s effects on the array it was called with, after each call to `fill_random`, print a message to the screen saying how many rows of X's in what interval of lengths you should see, and then call `bar_chart` for the array filled by `fill_random` -- that should allow you to see visually what ranges of values were put into the array by `fill_random`.
- (remember to list ALL the .cpp files for functions involved in `fill_random_test.cpp` when you try to compile that program...)

Submit your files `fill_random.h`, `fill_random.cpp`, and `fill_random_test.cpp`.